Table of Contents

[USEFUL LINKS 1](#_Toc494561068)

[INSTALLING ANGULARJS IN YOUR APP/PROJECT 2](#_Toc494561069)

[ABOUT ANGULAR 2](#_Toc494561070)

[FILES/FOLDER FORMAT 3](#_Toc494561071)

[THE BASICS 3](#_Toc494561072)

[INDIVIDUAL PARTS/FILES OF ANGULARJS AND FORMAT EXAMPLES 4](#_Toc494561073)

[WORKFLOW 6](#_Toc494561074)

[DIFFERENT DEPENDENT MODULES AND BOWER COMPONENTS COMMONLY USED IN MY ANGULARJS APPS 8](#_Toc494561075)

[DIFFERENT SERVICES I USUALLY USE IN MY FACTORIES AND CONTROLLERS 8](#_Toc494561076)

[THE DIFFERENCE BETWEEN USING AND NOT USING $SCOPE WITHIN THE CONTROLLERS 10](#_Toc494561077)

[ANGULARJS DIRECTIVE BASICS AND EXAMPLES 12](#_Toc494561078)

[ANGULARJS FILTERS 13](#_Toc494561079)

[USING FILTERS TO DYNAMICALLY SEARCH/SORT A TABLE OR ARRAY (SEARCH BOX OR BUTTON) 15](#_Toc494561080)

# USEFUL LINKS

* Angular tutorial through w3schools - <https://www.w3schools.com/angular/>
* Angular directives reference - <https://www.w3schools.com/angular/angular_ref_directives.asp>
* Angular documentation and main site - <https://angularjs.org/>
* AngularJS services - <https://docs.angularjs.org/guide/services>
* Angular factory vs service vs provider - <https://tylermcginnis.com/angularjs-factory-vs-service-vs-provider/>
* Angular 1 fundamentals part 1 - <https://scotch.io/tutorials/angularjs-1-x-fundamentals-part-1>
* Testing AngularJS with Jasmine and Karma part 1 (TDD) - <https://scotch.io/tutorials/testing-angularjs-with-jasmine-and-karma-part-1>

# INSTALLING ANGULARJS IN YOUR APP/PROJECT

* Before you install any angular or bower components, start by creating a bower.json file to keep track of your dependencies. This way, even if you delete the modules, you can just type **bower install** and bower will re-install everything.
* **IMPORTANTE NOTE:** This may or may not be necessary but due to how I set up bower the first time, currently in order to install anything, I need to type “sudo” before the command and “--allow-root” after it, ex: **sudo bower init-y --allow-root**, **sudo bower install --save angular --allow-root**
* To start, type **bower init -y**, just hit enter repeatedly and Y at the end
* Now you install your individual bower components, you may not need them all, but you ALWAYS add “--save” inbetween the install and name of your component when you type the command to install your bower component. If you don’t save it, it won’t be added to the bower.json file, and won’t be installed later if you just type “bower install”
* Start by typing **bower install --save angular**, you will always need this. These next ones you may or may not need, the most common ones I use are:

**bower install --save angular-route**

**bower install --save angular-cookies**

**bower install --save angular-messages**

* Lastly, make sure that you add the link/script to your index.html files, once again, you may not need them all, also be sure to add them to your angular module in your app.js file, or whatever you name it.

**<script src ='angular/angular.js'></script>**

**<script src="angular-route/angular-route.js"></script>**

**<script src ='angular-cookies/angular-cookies.js'></script>**

**<script src ='angular-messages/angular-messages.js'></script>**

# ABOUT ANGULAR

* AngularJS is a JavaScript framework/library that can be added to an html page. It extends HTML attributes with “directives” and binds data to html with “expressions”.
* AngularJS is an MVC framework on its own, with the index/partials being the view, and model/controller being implemented in JavaScript. The model and controller can differ from developer to developer, though the way I use it, the controller is obviously the controller, and the model is the factory, which I use to handle the logic, pre-server validations, query work, etc. The model is usually a service (which is what a factory is), but different people prefer different services.
* Angular models define angularjs apps, the controllers control it. The **ng-app** directive defines the application, the ng-controller directive defines the controller.
* Angular uses services, a service is a substitutable object that is wired together using dependency injection (DI), and is used to organize and share code across an app. For example, $http is a service. However, it is useful to create your own, so I most commonly do this with a factory, which is a service. A factory is also an object that you add properties to, then return it, and pass it to the controller.

# FILES/FOLDER FORMAT

* **GENERAL STRUCTURE:** All of the client side files (partials, angular stuff, etc.) is kept in the client folder. Within it, for a standard single page app, you have index.html, the controllers folder, the factories folder, the partials folder, the static folder, and the file that creates you angular module and routing, usually called “app.js”

# THE BASICS

* AngularJS extends HTML with **ng-directives**, some common and important ones are: **ng-app**, which goes at the top of index.html and defines an angularJS application, **ng-model** which binds the value of html controls (input, select, textarea) to application data, and **ng-bind** which binds application data to the HTML view. All directives start with “**ng-**”
* Angular expressions are written within html code in double curly brackets. Ex: if you have in your html page: **<h1>{{ 5+5 }}</h1>**, then 10 would be written as a header 1. But expressions can also be written inside a directive, ex: **ng-bind=“expression”**
* **JavaScript Expressions VS AngularJS Expressions:** Both can contain literals, operators, and variables. But AngularJS can’t be written inside HTML, nor can you use conditionsl, loops, and exceptions with angularjs.
* **ANGULARJS CONTROLLERS:** Are javascript objects created by a standard javascript object constructor. You define their application or scope with **ng-controller**, meaning that if I want a partial to use a certain controller, I would wrap the entire partial in say a div, and write **<div ng-controller= ‘controller\_name\_here’>**.
* **ANGULARJS FACTORIES AND SERVICES**: A service is a substitutable object that is wired together using dependency injection (DI), and is used to organize and share code across an app. A factory is an AngularJS service, and an object that you add properties to, then return the object, and pass it to the controller as service. Factories are one of the most popular services, and the one I use with angular and full mean to communicate with the client side controller and server side controller. This is usually within the factories folder and is where I put my http requests, logic, pre-server side validations, and query work. Any variables created in a factory are private, you would need to pass them to a function (like I do below), or write getter/setter methods. You need to include a factory’s name in the controller function with the other services for that controller to have access to it.

# INDIVIDUAL PARTS/FILES OF ANGULARJS AND FORMAT EXAMPLES

* **APP/MODULE FILE:** Usually called app.js, this is the file that you create and name your angularjs module. This is also where you write your angular routes and list your dependent modules. The dependent modules are used throughout your application and if they are not listed here, you don’t have access to them. All bower components are listed here as dependent modules, ex: angular-routes is ngRoute, and angular-cookies is ngCookies. See the format example below in red, the name of your variable “app” in this case, is the name of your angularjs module. It goes in front of the .config for your routes, your controllers, and your factories (as shown in the examples below. The first parameter in your module, “myApp” in this case, is the html element that your application goes within, it basically defines the scope of your entire angularjs application, it goes in the body or more commonly, in the head of the html file, for example “**<html ng-app= ‘myApp’>**”. After the html element is an array, inside is where you put your dependent modules, really any bower components you install (listed in the installing section above). Finally, you start your angular routing section with “the\_name\_of\_your\_module.config( function( $routeProvider){ routes here….}”. You then start the routes with “$routeProvider”, below that you list each route with “**.when(‘route\_here’, {templateUrl: ‘file\_path\_here’})**, the last thing you usually put is a “.otherwise()”, which the redirect route inside the parenthesis.

var app = angular.module( 'myApp', [ 'ngRoute' ])

app.config(function($routeProvider){

$routeProvider

.when('/',{

templateUrl: 'partials/main.html'

})

.when('/example',{

templateUrl: 'partials/userList.html'

})

.otherwise('/')

})

* **STILL APP/MODULE FILE:** Also, when you’re writing your angular routes, you could define the controller and it’s alias for each route, as shown below, this is completely optional and I usually do that in the actual partial file, see below:

var app = angular.module('app', ['ngRoute']);

app.config(function($routeProvider) {

$routeProvider

.when('/', {

templateUrl: 'partials/log\_reg.html',

controller: 'logregController',

controllerAs: 'logregCtrlr'

})

.when('/home', {

templateUrl: 'partials/home.html',

controller: 'homeController',

controllerAs: 'homeCtrlr'

})

.otherwise({

redirectTo: '/'

});

});

* **FACTORIES:** Creating a factory is very similar in format to a controller. You start with “module\_name.factory( ‘factory\_name’, function(){…})”, you can name your factory whatever you want, it is followed by a function, with any other services listed inside. Other services I commonly use in my factories are $http, $routeParams, $location, $cookies, these are all optional but if they aren’t listed here, you can’t use them in your factory. Inside your factory you always create an empty object, which represents your factory, and at the end, you always return that object. If you create any variables inside your factory with ‘var name =’, then that variable is private and you can’t access it inside your controller unless you write getter/setter methods. If you want your controller to have direct access to something, you create them as properties to your factory object, for example: “object\_name.str = ‘test’ ”, “object\_name.func = function(){}”. See the example below in blue:

app.factory( 'factory\_name\_here', function( $http, $location ){

var object\_name = {}

object\_name.example = 'Example here'

object\_name.show = function(callback){

callback(users);

}

return object\_name

})

* **CONTROLLERS:** A similar format with factories, a controller is started with “module\_name.controller( ‘controller\_name’, function(factory\_name, services){..}”, you can see the difference between using and not using $scope in a section below, this example won’t be using $scope. You don’t need to create or return an object with a controller. When you’re not using $scope, the first thing you do inside your controller is “var self = this”, the reason why is also listed in the using/not using $scope section below. You create any variables or functions with “this.var =”, you also call any factory functions or variables with “factory\_name.variable” or “factory\_name.function()”. Lastly, if you call a factory function inside your controller, but not inside a function, that factory name is automatically called and executed when the controller is loaded. See the format example below in green:

app.controller('first\_controller', function(first\_factory, $location){

var self = this;

this.test = 'test'

first\_factory.show(function(data){

self.users = data;

})

this.create = function(){

first\_factory.create(this.newUser);

$location.url('/partial2'); /

}

this.delete = function(index){

first\_factory.delete(index);

}

})

* **IMPORTANT NOTE/POINTER FOR FACTORIES AND CONTROLLERS:** As shown above, both controller and factory formats are very similar, but one useful thing I let about above was that you can create shorter names for your services, this especially comes in handy in larger, full mean apps, where you type them more often. You do this by, putting both the function and the services in an array. You list the services (and factory name if you’re making the controller) in quote marks, followed by the usual function. Inside the function, you list the new short names, in the same order as you list the services, that way the controller/factory knows which short name is for which service. ex: “([ ‘$location’, ‘$cookies’, function(location, cook){ } ]). See the factory example below in blue and controller in green:

app.factory('factory\_name', ['$http', '$routeParams', '$location', '$cookies', function(http, routeP, location, cookie){

var factory = {}

code here....

return factory

}])

app.controller('controller\_name', ['factory\_name', '$location', '$cookies', function(fact, loc, cook){

code here....

}])

# WORKFLOW

* **Note:** This and the full mean workflow is also available in the word doc: mean\_basics\_and\_steps\_for\_a\_refresher
* **IMPORTANT NOTE FOR WHEN YOU VISIT THE WEBSITE/APPLICATION:** The important part to understand is what happens first. Say you go to <http://localhost:5000/>, ordinarily a routes.js file or its equivalent would tell you where to go for the ‘/’ or root route (ex: in a mean application “app.get(‘/’, function….)” would do it). And if you have that in your routes.js file and have that loaded before your client folder, then that will take precedence and tell it what to do. But normally, you don’t have that in your routes.js file (standard for my apps), and that file is loaded after your client folder. In this instance, you’re loading up the application but not telling node what to do for an empty/root route or the “/” route. So, like many webservers, express will automatically load “index.html” for the root route if it’s not told what to do. So, in a standard single page MEAN application, you hit the site and express loads the file called “index.html”, if you don’t have that, then nothing gets loaded and you have a blank screen.
* **WHEN YOU VISIT THE WEBSITE/APPLICATION - INITIAL LOADING:** **First**, as stated in the bullet point above, when you hit the app/website, express loads index.html. **Second**, once index is loaded, it starts reading the file and loads all of your stylesheets and bower components first (if you try to load angular things before the bower components, it will not work). The next big step is whenever it hits the angular files. These files should be loaded in the proper order since the factory requires the angular module, and the controller requires the module and usually a factory. So, the **third** big step is whenever it loads the file containing your angular module, usually app.js. This is where you load the major angular components/modules, and where you set up your angular routing. The **fourth** important thing happens is after it finishes with your app.js file, it loads first your factories then your controllers. Now, the **fifth** important part is **when angular routing kicks in**. I’m not certain on when exactly it kicks in, but it is most likely after all of the files in the header of index.html gets loaded and it hits the **ng-view** in the body. It goes to the app.js file to load a partial, and when you first hit the site, it either loads the ‘/’ route, or where the .otherwise tells it to go if the ‘/’ route isn’t listed. this tells it what partial to initially load within the ng-view.
* **THE REGULAR WORKFLOW OF JUST ANGULARJS:** **First**, the view does something, say you click a button to create a user and it triggers a function in the controller. **Second**, you are sent to the controller from there, you could be sent back to the view but usually, the **third** step is then being sent to factory via a callback function. From the factory many things could happen, it could go back to the controller, or the **fourth** step is being sent to the server via an $http request (usually with json data attached). After the server does its thing, it sends back a json response. The **fifth** step is getting that response and either doing something to it or passing it straight back to the controller. The **sixth** and final step is what happens in the controller, whether it does anything to it, or just sends it straight to the view.
* **THE REGULAR WORKFLOW OF FULL MEAN WHEN USING THE APP:** So, after the startup, you are in your app with a partial loaded. Say you hit a button to load data from the database. **First**, is the action that makes it hit the controller, whether it be the loading of a partial or a button/form. **Second**, you go to the controller, which does its thing and can stop from there, but it usually hits the **third** step and sends you to the factory via a function usually. This is where the logic happens, once again, it can stop here, but if you need to hit the database, then the **fourth** step is when the factory makes an $http request (usually with some json data) and sends you to the server. The **fifth** step is after making the $http request, you are sent to the router.js file in /server/config, which tells you what to do for each route. The **sixth** step is what the router.js file tells you to do from there, which is ordinarily to go to a certain method within a certain controller. Once you are in the controller, usually the **seventh** step is to hit the model and mongoDB via a query and either callbacks or promises. The **eight** step is the response you get, once the query is done, the controller sends back a response in json format. The **ninth** and final step is sending the data back, first to the factory, then to the controller, and finally back to the view. Most likely the factory and controller would do something to it, but this is a simplified verions.

# DIFFERENT DEPENDENT MODULES AND BOWER COMPONENTS COMMONLY USED IN MY ANGULARJS APPS

* **ANGULAR COOKES - ngCookies:** The cookies provider or module allows read/write access to the browser’s cookies. A cookie is a text file on a computer stored by a browser for a specific website. Basically a front end session, they are commonly used for authentication, storing of site preferences, shopping cart items, and server session identification, and normally, each cookie only works for a specific site. You use **$cookies.get(‘key’)** to retrieve the value, and **$cookies.put(‘key’, value)** to set the value of a cookie, and **$cookies.remove(‘key’)** to remove/destroy a cookie. For example: **$cookies.put(‘test’, ‘Hey there’)**, this creates a cookie called ‘test’ and makes its value ‘Hey there’. And doing **console.log( $cookies.get(‘test’) )** would log “Hey there” to the console, and doing **$cookies.remove(‘test’)** would get rid of it.

# DIFFERENT SERVICES I USUALLY USE IN MY FACTORIES AND CONTROLLERS

* **$ROUTEPARAMS:** The $routeParams service allows you to retrieve the current set of route parameters. Requires ngRoute (angular-route) to be installed and included in your angularjs module (app.js file). Basically, any paramters you send through a route can be accessed with $routeParams. For example, let’s say I hit the user show route for a partial, or **“/user/:id**” as it would be shown in your angular routes file. So, whenever I go to this route, if I did **console.log($routeParams)** in the controller or factory, this would be what it would look like: **{id: "59cbd11929326749fb0cb8d0"}**, so in order to access any parameters, you access them like they are keys to the $routeParams object. Meaning, I could access that id with: **$routeParams.id**, which would access the key “id” in $routeParams. Now, for the SERVER part of it, you no longer use $routeParams, but they go hand in hand so it’s useful to note here. Let’s say I did an http request to the back at the route “**/appointments/edit/:id**”, then in order to access it in the server controller, I would do “**req.params.id**”, basically you use the first parameter for you controller method, usually “request” or “req”, then followed by “params.param\_name”
* **$HTTP:** The http service is critical to how the client and server parts of full mean talk to one another. Via the $http service, I can send an http request from wherever I want on the client side (usually the factory) to hit a route in the routes.js file on the server. You can only send your data/response in json format or as a javascript object. The http request is asynchronous, so you use usually use a callback, the format is as follows: **$http.post(‘/login’, data).then(function(response){….}**, you could change the .post, that is simply the action you’re making (get, post, etc.). Inisde the parenthesis is the route to hit, then it is any data you want to send over, this can only be json format and is optional. You could put the entire object there, but I usually put it into a variable first. It is followed by a .then(), from there things can be a little different. You could do a .then(sucessCallback, errorCallback), but I usually just use one function and if statements based on my response. “response” is just what I called it, but the function needs a parameter, which represents the response from the server. This response is an object with multiple keys that are irrelevant, the only thing we need is the “response.data” key, this contains the actual response that we meant to send over from the server, as an object. Below is a factory method that contains an http request, the method gets a “user” which is equal to an object of the name, email, and password, and a callback function. This app is simple, so on the server I just send over {message: true} if everything was good or {message: false} if it wasn’t. See below:

factory.login= function(user, callback){

$http.post('/login', user).then(function(response){

if(response.data.message){

callback(true)

} else {

callback(false)

console.log('not great')

}

})

}

* **STILL $HTTP:** That is a basic version of the http request from the client side. Just like the client side, the server side can only send over json data, so the server does with the **res.json( {data…})** method. Where “res” is the response, I simply called the parameter res (you’ll see below), and json only accepts an object as a parameter/value. So, I usually pick specific keys, give them specific data, and do something accordingly on the client side. See below what happens on the controller side for the login:

login: function(req, res){

LoginReg.findOne({email:req.body.email}, function(err, result){

if(err){

res.json({message: false});

} else if(!result){

res.json({message: false});

} else {

var check = result.validPassword(req.body.password)

if(check){

console.log('good login');

res.json({message: true});

} else{

console.log('unsuccessful login');

res.json({message: false});

}

}

})

}

* **$LOCATION:** The $location service is basically like a redirect, it tells angular to go to this partial route. For example, if I put **$location.url(‘/users’)**, then it would make angular go to the partial route: **localhost:8000/#/users**

# THE DIFFERENCE BETWEEN USING AND NOT USING $SCOPE WITHIN THE CONTROLLERS

* **$SCOPE:** is the application object or the owner of the application (controller’s) variables and functions. Basically, when used in a controller, it has access to and controls that controller’s variables and functions/methods.
* **HOW TO USE $SCOPE:** To use it, when creating a controller, you pass it to the controller function as a parameter, ex: **app.controller(‘NameHere’, function($scope){…..})**. Afterwards, any variables or functions in that controller need to have “$scope.” in front of them, or you won’t be able to access them in the view and possibly other parts of the controller, ex: $scope.name = “Jack”, $scope.test = function(..){…}. In the view, you don’t refer to these variables or methods with scope in front of them, for example, I would access the name variable above with **{{name}}**. Not sure how this would work if multiple controllers are used in the same partial and had a variable or method of the same name.
* **USING $SCOPE:** As shown above, you pass in “$scope” as a parameter to the controller function. Then, in order to use anything created in your controller in the view, you need to add “$scope.” to the front. A benefit to this is that you can reference that variable directly in the view, unlike when you’re not using “$scope” and you need to write the name or short name of the controller ahead of it (shown in the not using $scope bullet). See the controller example in green, and the partial in blue:

app.controller('$scope\_example\_controller', function(first\_factory, $scope){

$scope.name = 'Jack\_2\_ng-model\_example\_using\_scope'

$scope.arr = ['Hey', 'There', 4]

first\_factory.show(function(data){

$scope.users = data;

})

})

<div ng-controller='$scope\_example\_controller'>

<h1>All Users</h1>

<table>

<tr>

<th>First Name</th><th>Last Name</th><th>Fav Language</th><th>Actions</th>

</tr>

<tr ng-repeat='user in users'>

<td>{{user.first}}</td><td>{{user.last}}</td><td>{{user.fav}}</td>

</tr>

</table>

<form class="" action="index.html" method="post">

<input ng-model="name">

</form>

<h3>This partial and controller was made to demonstrate the difference between using $scope or not within the controller.</h3>

<p>How you show a variable with scope: The array 'arr' is: {{arr}}</p>

</div>

* **STILL USING $SCOPE:** As you can see, the benefit is referring the variables directly. See the next bullet to see the difference when you’re not using scope.
* **NOT USING $SCOPE:** Not using $scope focuses on using “this” and “self” instead. You don’t need to pass anything as a parameter to the controller function as a replacement for $scope. “**this**” is available in the controller function by default and refers to the controller and it’s variables, methods, etc.
* **STILL NOT USING $SCOPE, DIFFERENCE BETWEEN - SELF/THIS:** HOWEVER, an important note is how we use “**self**” in the controller. You can declare and access a variable in a controller like so “**this.name = ‘Jack’; console.log(this.name);”**, however, if you’re within a function that doesn’t belong to the controller, then the scope of “this” changes and refers to the object that owns the function. For example, see the controller below, and how I call the factories function “show” with “first\_factory.show(….)”. Inside the factory, the show function takes a callback function as a parameter and calls that callback function with an array of objects as a parameter. What’s important is that even though I write out what the callback function does with the array in the controller, it is called/activated within the factory, so when the function actually runs, it is within the factory. Therefore, if I write “this” in the function, “this” refers to the factory and not the controller. Notice the console.logs within the show’s callback function. doing “console.log(this.name)” shows undefined because there is no name variable in the factory. But “console.log(self.name)” shows the name variable from the controller. Basically, at the start of my controller I write “**var self = this;**”, this basically creates a variable and assigns the “this” that refers to the controller to it. You could name it anything, but another way to picture it is: “**var self = TheCurrentControllerWeAreIn;**”. To keep it simple, whenever we declare “self” like this, it means that no matter where we call it, it refers to the controller it was created in. See below, controller is green, comments are red, and the partial is blue:

app.controller('userListsController', function(first\_factory){

var self=this;

this.name = 'Jack\_1\_ng-model\_example\_not\_using\_scope'

first\_factory.show(function(data){

self.users = data;

console.log(this.name); //undefined, “this” refers to first\_factory

console.log(self.name); // 'Jack\_1\_ng-model\_example\_not\_using\_scope', no matter if we’re in the controller or first\_factory, “self” still refers to the controller.

})

})

<div ng-controller='userListsController as user\_controller'>

<h1>All Users</h1>

<table>

<tr>

<th>First Name</th><th>Last Name</th><th>Fav Language</th><th>Actions</th>

</tr>

<tr ng-repeat='user in user\_controller.users'>

<td>{{user.first}}</td><td>{{user.last}}</td><td>{{user.fav}}</td>

</tr>

</table>

<input ng-model="user\_controller.name">

<input type="submit" name="" value="Test" ng-click="user\_controller.test()">

</div>

* **STILL NOT USING $SCOPE:** The other important part about not using $scope is how you do things in the view/partial. Basically, we have to reference all of our variables with “**controllerName.**”, so since our controller in this example is named “userListsController”, if we wanted to show the “name” variable, we would have to write “{{**userListsController.name}}**”. This is a little long and annoying to type, so we can create a sort of nickname/short name to use instead. Whenever you write “ng-controller=”, followed by the name of the controller, we can write “as nickname” after it. This creates another name that we can use to reference that controller and it’s variables/methods. For example, if I write “**<div ng-contorller= ‘userListsController as users’>**”, then I could access the name with “**{{users.name}}**”, much easier to write.

# ANGULARJS DIRECTIVE BASICS AND EXAMPLES

* **NG-APP:** is where you define the root element of the angularjs application. This directive will “auto-bootstrap” (automatically initialize) the application when a web page is loaded. This can go at the start of your body but is better off at the start of your index.html file, ex: **<html ng-app='myApp'>**
* **NG-INIT:**is where you can basically declare variables or gives things values. For example, if I wrote **<div ng-init=“myCol= ‘lightblue’ “>**, then in another line wrote **<input style= “background-color: {{myCol}}” ….>**, then it would read that as “background-color: ‘lightblue’”. Another example, you could declare two variables by doing **<div ng-init= “quantity=1; cost=5; name=‘John’”>**, and use either variable as you want. Every directive, ng-init included, is like class in the sense that you put them in tags, ex: <div directive\_here>, <input directive\_here>, <p directive\_here>. You could also create objects or arrays using ng-init.
* **NG-MODEL:** This directive allows you to bind the value of an input field to a variable created in angularJS, for example, if in my controller I have **this.name = ‘Jack’** (not using $scope in this example), and then had an input like this: **Name: <input ng-model=“controllerName.name”>**, then this would show up on the page as an input with “Jack” written in it to start. This is a two way binding however, meaning that if I click that input and typed to change it to “Jack swanson”, then the value of “this.name” in the controller would also change, until I reloaded the page/partial, they are linked.
* **NG-REPEAT:** is a directive used to repeat an html element, similar to a for loop in javascript. For example, if I wrote **<div ng-init= “names=[‘Jani’, ‘Hege’, ‘Kai’]”>**, then wrote:

**<ul>**

**<li ng-repeat=“x in names”>**

**{{x}}**

**</li>**

**</ul>**

* **STILL NG-REPEAT:** Then it would show each index in the names array as a list item. You could also use this on an array of objects (like data you get from mongoDB sometimes). For example, if names was **[ {name: ‘Jani’, country: ‘Norway’}, {name: ‘Hege’, country: ‘Sweden’}, {name: ‘Kai’, country: ‘Denmark’} ]**. Then you would write the repeat code as:

**<ul>**

**<li ng-repeat=“x in names”>**

**{{ x.name + ‘, ’ + x.country }}**

**</li>**

**</ul>**

* **NG-SHOW & NG-HIDE:** These two directives are similar to ng-if, they are basically if statements that hide or show something based on how the condition evaluates. For example, let’s say we had a table filled with data from the db, and if the db was empty, the table would also be empty, so in that case we wanted a row filled with “ - ” to be shown instead, we could write: “**<tr ng-show= ‘ctrl.users.length == 0’> <td> - </td> </tr>**”, this basically says “if ctrl.users.length is equal to 0, then show this row”. If the length wasn’t 0, it wouldn’t be shown, so for ng-show, a true expression will be shown while a false expression won’t. ng-hide is opposite, a true expression will be hidden, or won’t be shown, while a false expression will be shown. So, if we added onto the above code with: “**<tr ng-hide=‘ctrl.users.length == 0> <td> data here, probably an ng-repeat… </td> </tr>**”, and that would only show if the length of ‘ctr.users’ wasn’t 0. I used this as an example, but using ng-hide in that instance wouldn’t be necessary, an ng-repeat would be more useful considering if the db was empty, there wouldn’t be an data to hide.
* **NG-BIND:** like it sounds, literally binds a value to something. If I wrote: **<p> The third result is <span ng-bind=“16”></span></p>**, then it would translate to “The third result is 16”.
* **CREATING DIRECTIVES**: I have yet find a need to create a new directive but you can with the **.directive** function, see how in this url: <https://www.w3schools.com/angular/angular_directives.asp>, in the middle of the page is where it shows it.

# ANGULARJS FILTERS

* Filters can be added to angularJS to format data, see this url for more info: <https://www.w3schools.com/angular/angular_filters.asp>. You add them to expressions with the | pipe character, ex: {{ lastName | uppercase}}.
* **NOTE**: In the examples below, I use “this” in the controller and refer to that controller in the view as “c”.
* **NOTE**: You can add an alias to the ng-repeat when you use filters, very important and useful, see the bullet below.
* The most common filters are listed below:

currency Format a number to a currency format.

date Format a date to a specified format.

filter Select a subset of items from an array.

json Format an object to a JSON string.

limitTo Limits an array/string, into a specified number of elements/characters.

lowercase Format a string to lower case.

number Format a number to a string.

orderBy Orders an array by an expression.

uppercase Format a string to upper case.

* **EXAMPLES - UPPERCASE, LOWERCASE, AND CURRENCY**: If you had a variable in the controller: “this.name = ‘Jack’ ”, then used lower/uppercase in the view, **{{ c.name | lowercase}}** would show “jack”, while **{{ c.name | uppercase}}** would show “JACK”. Also, if you had a variable: “this.num = 24”, then used the currency filter in the view like so: **{{ c.num | currency }},** then it would show “$24.00”
* **USING ORDERBY AND FILTER WITH NG-REPEAT AND AN ARRAY:** You can use filters with directives such as “ng-repeat” and these two work specifically with arrays (even an array of objects). For example, if I had an array: **this.names = [ {name: ‘Jani’, country: ‘Norway’}, {name: ‘Finn’, country: ‘Finland’}, {name: ‘Ben’, country: ‘USA’}, {name: ‘Sam’, country: ‘England’} ]**, and in the view wrote an ng-repeat like: **<li ng-repeat= “x in c.names | orderBy: ‘country’ ”> {{ x.name + ‘, ’ + x.country}} </li>**, then it would show all of the indexes orderby the name of the country, from a-z, so first it would show the England index, then Finland, then Norway, etc. If you want to order them in descending order you add a minus to the orderBy key, inside the parenthesis, for example: **<li ng-repeat= “x in c.names | orderBy: ‘-country’ ”>**. Now, the “filter” is similar to using “%i%” in sql. It only returns/shows the indexes that match the item(s). For example, if I have an array: **this.user = [ “ben’, ‘John’, ‘sam’, ‘susan’]**, then use **<li ng-repeat= “u in c.user | filter: ‘s’ ”> {{ u}} </li>**, then it would only show sam and susan, since they both have an ‘s’ character. It doesn’t matter if it’s lowercase, uppercase, or where it is located, just as long as it has an ‘s’. Going back to the names array above, which contains object, if I used filter like I just did, the filter would be run against every key in each object, only one key would need to pass for that index to be shown. so “ **| filter: ‘jani’** ” would only cause the first index to be shown, while “ **| filter: ‘usa’** ” would only cause the last index to be shown. Lastly, if you’re working with an array of objects and want to use filter on specific keys, you put an object for filter’s value. For example: **<li ng-repeat= “x in c.names | filter: {country: ‘f’} ”> {{ x}} </li>** would only cause filter to check the country key of each index and see if it contained an f, but I could expand on that with: **<li ng-repeat= “x in c.names | filter: {country: ‘f’, name: ‘finn’ } ”> {{ x}} </li>**, this would cause it to check if an index had ‘f’ in the country and a name that contained ‘finn’, which would cause only the third object to be shown.
* **USING ALIASES WITH NG-REPEAT AND FILTERS:** You can use an alias on an ng-repeat that is being filtered in order to reference the filtered results. For example, let’s say I had an array of users, and I used the ‘filter’ filter and ng-model to create a search box to a table showing the results (see section below if you’re unfamiliar with this). So the code would look like this:

<input type= “text” ng-model= “search\_box”>

<table>

<tr>

<th>user Name</th>

</tr>

<tr ng-repeat=‘user in c.users | filter: search\_box as filtered\_results’>

<td>{{ user }}</td>

</tr>

<tr ng-show= “filtered\_results.length == 0”>

<td> No users found </td>

</tr>

</table>

* **STILL USING ALIASES WITH NG-REPEAT AND FILTERS:** So, with the above code, once you type into the text input, the table would be dynamically filtered based on your results. But, if nothing matched, then the table below the head would be blank, no rows shown. So, we make an ‘ng-show’ that will show a default row if no users are found in the filter. By adding “as alias\_name\_here” after the filter, we basically create an alias or a variable that is equal to the filtered results of the ng-repeat. We can then use that with the ng-show to basically say “if the filtered ng-repeat doesn’t show anything, then show this”

# USING FILTERS TO DYNAMICALLY SEARCH/SORT A TABLE OR ARRAY (SEARCH BOX OR BUTTON)

* **USING FILTER, NG-REPEAT, AND AN INPUT (LIKE A SEARCH BOX FOR A TABLE):** If we had a simple array like: “**this.names = [ ‘Jack’, ‘Ben’, ‘Turner’, ‘Jimmy’, ‘Sam’, ‘Leo’]**”, as shown in the bullet point above, we could use the “filter” filter to determine which names showed up in an ng-repeat. However, we could do this dynamically with ng-model. Since the ng-model directive has two way binding, if we tied ng-model to a text input, then made filter go by that ng-model, we could dynamically determine what gets filter by typing in the input, similar to a search bar in a table. Whenever you put something in as the filter value without quotes, it reads it as a variable. For example, see the html below in green:

<p> <input type= ‘text’ ng-model=“test”> </p>

<ul>

<li ng-repeat= “x in c.names | filter: test”>

{{ x }}

</li>

</ul>

* **USING NG-CLICK AND ORDERBY TO DYNAMICALLY ORDER A TABLE OR ARRAY VIA A BUTTON:** Similar to the above method, if have an array of objects and show them via a table, you can use ng-click and orderBy to dynamically order the table. Say I have this array: **this.names = [** **{name:'Jani',country:'Norway'}, {name:'Carl',country:'Sweden'}, {name:'Margareth',country:'England'}, {name:'Hege',country:'Norway'}, {name:'Joe',country:'Denmark'}]**, now, when we write the ng-repeat we use orderBy and give it a variable, ex: **ng-repeat= “x in c.names | orderBy: what\_to\_order\_by”**, which when the page is loaded could be blank or we would give it a default value with **ng-init= “what\_to\_order\_by= ‘name’ ”**, even if we didn’t the variable is created as soon as we give it to orderBy, it would just be blank starting out. Now, when we write our table, we could use ng-click to assign/change the value of “what\_to\_order\_by”, ex: **<th ng-click= “what\_to\_order\_by(‘name’)”>Name</th>**. This way, by clicking the header row for each column, we could dynamically change what “what\_to\_order\_by” is equal to and therefore, change how the table is ordered. We could even make another button on its own to determine if it is sorting descending or ascending. See my code below for a basic example:

<table>

<tr>

<th ng-click= “what\_to\_order\_by(‘name’)”> Name </th>

<th ng-click= “what\_to\_order\_by(‘country’)”> Country </th>

</tr>

<tr ng-repeat= “x in c.names | orderBy: what\_to\_order\_by”>

<td> {{ x.name }} </td>

<td> {{ x.country }} </td>

</tr>

</table>